Problem Set 1 **vgs3qt**

data structures and algorithms 2 - fall 2023 due september 8, 2023 at 11:59 pm

Collaborators: Yining Liu(yl2nr)
Sources:

problem 1 *Order class proof using limits*

Prove that $n^k \in o(a^n)$ for $a > 1$. Use the limit definition of the order class. If you find you need to use L'Hôpital's rule, note that $(a^n)' = (\log a)a^n$.

Solution:

Proof.

We want to show $\exists m > 0$ s.t. $\forall n > k$, $n^k < a^n$. This means $\lim_{n\to\infty} \frac{n^k}{a^n} = 0$ as by the definition of a sequence being convergence. If $\lim_{n\to\infty} \frac{n^k}{a^n} = 0$, then we can get that $\exists m > 0$ s.t. $\forall n > m$, $\frac{n^k}{a^n} < 1$ thus $n^k < a^n$.

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{\frac{dn^k}{dn}}{\frac{da^n}{dn}} = \lim_{n\to\infty} \frac{kn^k}{(loga)a^n}$$

Since both nominator and denominator approach 0, we can apply L'Hôpital's rule. By repeatedly applying L'Hôpital's rule k times and make the nominator becomes a constant.

$$\lim_{n\to\infty} \frac{n^k}{a^n} = \lim_{n\to\infty} \frac{\frac{dn^k}{dn}}{\frac{da^n}{dn}} = \lim_{n\to\infty} \frac{kn^k}{(loga)a^n} = \lim_{n\to\infty} \frac{\frac{dkn^k}{dn}}{\frac{d(loga)a^n}{dn}} = ... = \lim_{n\to\infty} \frac{k!}{(loga)^k a^n} = \frac{k!}{(loga)^k} \cdot \lim_{n\to\infty} \frac{1}{a^n} = 0$$

Now we show that $\lim_{n\to\infty} \frac{n^k}{a^n} = 0$ and using the definition of convergence listed above, we prove that after certain values of n, n^k grows strictly slower than a^n for $a > 1$. □

PROBLEM 2 *FlyMe Airlines*

An airline, FlyMe Airlines, is analyzing their network of airport connections. They have a graph $G = (A, E)$ that represents the set of airports A and their flight connections E between them. They define $hops(a_i, a_k)$ to be the smallest number of flight connections between two airports. They define $maxHops(a_i)$ to be the number of hops to the airport that is farthest from a_i, i.e. $maxHops(a_i) = max(hops(a_i, a_j))\,\forall a_j \in A$.

The airline wishes to define one or more of their airports to be "Core 1 airports." Each Core 1 airport a_i will have a value of $maxHops(a_i)$ that is no larger than any other airport. You can think of the Core 1 airports as being "in the middle" of FlyMe Airlines' airport network. The worst flight from a Core 1 airport (where "worst" means having a large number of connections) is the same or better than any other airport's worst flight connection (i.e. its $maxHops()$ value).

They also define "Core 2 airports" to be the set of airports that have a $maxHops()$ value that is just 1 more than that of the Core 1 airports. (Why do they care about all this? Delays at Core 1 or Core 2 airports may have big effects on the overall network performance.)

Your problem: Describe an algorithm that finds the set of Core 1 airports and the set of Core 2 airports. Give its time-complexity. The input is $G = (A, E)$, an undirected and unweighted graph, where $e = (a_i, a_j) \in E$ means that there is a flight between a_i and a_j. Base your algorithm design on algorithms we have studied in this unit of the course.

Solution:

We need to run BFS from every vertex (airport) to find the corresponding *maxHops()*. When we traverse all the airports, we also memorize the least and second least value of *maxHops()*. Return the Core 1 airports and Core 2 airports based on the value remembered when we finish running all the BFS from every vertex (airports). Eg:

```
public class CoreAirportsFinder {
    public static Map<String, List<Integer>> findCoreAirports(A, E) {
        int minimum = 0;
        int secondLeast = 0;
        List<Integer> core1 = new ArrayList<>();
        List<Integer> core2 = new ArrayList<>();

        for (Integer a : A) {
            int maxHop = maxHopBFS(a, A, E);
            if (maxHop < minimum) {
                secondLeast = minimum;
                minimum = maxHop;
                core2 = new ArrayList<>(core1);
                core1 = new ArrayList<>();
                core1.add(a);
                }
            else if (maxHop == minimum) {
                core1.add(a);
                }
            else if (maxHop < secondLeast) {
                secondLeast = maxHop;
                core2 = new ArrayList<>();
                core2.add(a);
                }
            else if (maxHop == secondLeast) {
                core2.add(a);
                }
        }

        Map<String, List<Integer>> result = new HashMap<>();
        result.put("core1 airports", core1);
        result.put("core2 airports", core2);

        return result;
    }

    public static int maxHopBFS(start, A, E) {
        int maxHop = 0;
        Queue<Integer> toVisit = new LinkedList<>();
        int[] hops = new int[A.size()];
        Arrays.fill(hops, -1);
        boolean[] visited = new boolean[A.size()];
        visited[start] = true;
        hops[start] = 0;
        toVisit.add(start);

        while (!toVisit.isEmpty()) {
            int current = toVisit;
            List<Integer> neighbors = E.get(current);
            for (int node : neighbors) {
                if (!visited[node]) {
                    toVisit.add(node);
                    visited[node] = true;
                    hops[node] = hops[current] + 1;
                    maxHop = Math.max(hops[node], maxHop);
                }
            }
        }
        return maxHop;
    }
}
```

The time complexity (for each BFS) is $O(A + E)$.
Total complexity is $O(A^2 + A \cdot E)$.

PROBLEM 3 *Counting Shortest Paths*

Given a graph $G = (V, E)$, and a starting node s, let $\ell(s, t)$ be the length of the shortest path in terms of number of edges between s and t. Give a clear description of an algorithm that computes the number of distinct paths from s to t that have length exactly $\ell(s, t)$.

Solution:

We can use BFS and use a variable to count the number of shortest paths from s to t that have the length of $\ell(s, t)$. Eg:

```
public class ShortestPathCounter {
    public static int numShortestPath(G, s, t) {
        int[] depth = new int[G.size()];
        int[] numPath = new int[G.size()];
        boolean[] visited = new boolean[G.size()];
        Queue<Integer> toVisit = new LinkedList<>();

        Arrays.fill(depth, 1000);
        Arrays.fill(numPath, 0);
        Arrays.fill(visited, false);

        depth[s] = 0;
        numPath[s] = 1;
        visited[s] = true;
        toVisit.add(s);

        while (!toVisit.isEmpty()) {
            int current = toVisit.poll();
            int layer = depth[current];
            int numOfPath = numPath[current];

            for (int node : G.get(current)) {
                if (!visited[node]) {
                    toVisit.add(node);
                    visited[node] = true;

                    if (layer + 1 < depth[node]) {
                        depth[node] = layer + 1;
                        numPath[node] = numOfPath;
                    }
                    else if (layer + 1 == depth[node]) {
                        numPath[node] += numOfPath;
                    }
                }

            }
        }
        return numPath[t];
    }
}
```

PROBLEM 4 *Coffee*

Charlottesville is known for some of its locally-roasted coffees, each with its own unique flavor combination. Suppose a group of coffee enthusiasts are given n samples $c_1, c_2, ..., c_n$ of freshly-brewed Charlottesville coffee by a coffee-skeptic. Each sample is either a *Milli* roast or a *Shenandoah Joe* roast. The enthusiasts are given each pair (c_i, c_j) of coffees to taste, and they must collectively decide whether: (a) both are the same brand of coffee, (b) they are from different brands, or (c) they cannot agree (i.e., they are unsure whether the coffees are the same or different). Note: all n^2 pairings are tested, including $(c_i, c_i), (c_i, c_j)$, and (c_j, c_i), but not all pairs have "same" or "different" decisions.

At the end of the tasting, suppose the coffee enthusiasts have made m judgments of "same" or "different." Give an algorithm that takes these m judgments and determines whether they are *consistent*. The m judgments are *consistent* if there is a way to label each coffee sample c_i as either *Shenandoah Joe* or *Milli* such that for every taste-comparison (c_i, c_j) labeled "same," both c_i and c_j have the same label, and for every taste-comparison labeled "different," both c_i and c_j are labeled differently. Your algorithm should run in $O(m + n)$ time. Prove the correctness and running time of your algorithm. Note: you do *not* need to determine the brand of each sample c_i, only whether the coffee enthusiasts are consistent in their labelings.

Solution:

1. Create a hashmap (Java) or dictionary (Python) to memorize all the vertexes and edges given.

2. Then we go through the scenarios (m judgments). If judgment (c_i, c_j) is labeled as SAME, then:

If both ci, cj are not keys of the Hashmap, add them to the Hashmap like the vertexes in the graph.

If one of the vertex is in the Hashmap, then just add the other one in as a key into the Hashmap. Like a vertex in the graph without any edge.

If both are already in the keys of the Hashmap, meaning the vertexes are created already, then do nothing.

3. If judgment (c_i, c_j) is labeled DIFFERENT, then:

If both c_i, c_j are not created in the Hashmap, create them, and add an edge between them. So there will be an edge between (c_i, c_j) and (c_j, c_i).

If only one of c_i, c_j is not created in the key of the Hashmap, create it and also connect the edge similar to the prior condition.

If both c_i, c_j are the key of the Hashmap, just connect the edge between them like above.

4. If the judgment (c_i, c_j) is labeled CANNOT AGREE, then just not create any key in the Hashmap. Ignore them so we don't count any extra keys and edges.

5. After finishing all the judgments given (m judgments), we use BFS to check if the graph is Bipartite. If it is, then it means m judgments are *consistent*. Otherwise, it is *inconsistent*.

The time complexity of all the adding of keys is $O(1)$, so having m judgments and adding them results in a time complexity of $O(m)$. The BFS has a time complexity of $O(m + n)$ because there can be n vertexes and m pairs, thus m edges.

The overall time complexity is $O(2m + n) \Rightarrow O(m + n)$.

PROBLEM 5 *Ancient Population Study*

Historians are studying the population of the ancient civilization of *Algorithmica*. Unfortunately, they have only uncovered incomplete information about the people who lived there during Algorithmica's most important century. While they do not have the exact year of birth or year of death for these people, they have a large number of possible facts from ancient records that say when a person lived relative to when another person lived.

These possible facts fall into two forms:

- The first states that one person died before the another person was born.
- The second states that their life spans overlapped, at least partially.

The Algorithmica historians need your help to answer the following questions. First, is the large collection of uncovered possible facts internally consistent? This means that a set of people could have lived with birth and death years that are consistent with all the possible facts they've uncovered. (The ancient records *may not be accurate*, meaning all the facts taken together cannot possibly be true.) Second, if the facts are consistent, find a sequence of birth and death years for all the people in the set such that all the facts simultaneously hold. (Examples are given below.)

We'll denote the n people as $P_1, P_2, \ldots, P_n$. For each person P_i, their birth-year will be b_i and their death-year will be d_i. (Again, for this problem we do not know and cannot find the exact numeric year value for these.)

The possible facts (input) for this problem will be a list of relationships between two people, in one of two forms:

- P_i *prec* P_j (indicates P_i died before P_j was born)
- P_i *overlaps* P_j (indicates their life spans overlapped)

If this list of possible facts is not consistent, your algorithm will return "not consistent". Otherwise, it will return a possible sequence of birth and death years that is consistent with these facts.

Here are some examples:

- The following facts about $n = 3$ people are **not** consistent: P_1 *prec* P_2, P_2 *prec* P_3, and P_3 *prec* P_1.
- The following facts about $n = 3$ people **are** consistent: P_1 *prec* P_2 and P_2 *overlaps* P_3. Here are two possible sequences of birth and death years:
  $b_1, d_1, b_2, b_3, d_2, d_3$
  $b_1, d_1, b_3, b_2, d_2, d_3$
  (Your solution only needs to find one of any of the possible sequences.)

Your answer should include the following. Clearly and precisely explain the graph you'll create to solve this problem, including what the nodes and edges will be in the graph. Explain how you'll use one or more of the algorithms we've studied to solve this graph problem, and explain why this leads to a correct answer. Finally, give the time-complexity of your solution.

Solution:

I think we can use DFS and the topological sort to solve this question. We build a directed graph that has each person's death d_i as a vertex and birth b_i as another vertex. There should not be any cycles or backward edges if the graph is *consistent*.

There are two types of inputs:

1. If P_i *prec* P_j, meaning P_i died before P_j was born, we created four corresponding vertexes b_i, d_i, b_j, d_j if they are not already in the graph. Then we add edges to connect $(b_i, d_i), (d_i, b_j), (b_j, d_j)$.

So, we connect i's birth to his death and connect the death of the precedent to the birth of the decedent.

2. If *P_i overlaps P_j*. Their lifespan overlaps, so we add edges $(b_i, b_j), (b_j, d_i), (d_i, d_j)$. Here, we assume i is born first and dies first while j is alive when i die (one of the possibilities). So we connect i's birth to the j's birth and i's death to the j's death.

Then we apply DFS and Tropological sort to get the sequence of birth and death to check if it is consistent. Eg:

```
public class TopologicalSort {
    public static List<Integer> topSort(V, E) {
        int numVertices = V.size();
        boolean[] visited = new boolean[numVertices];
        List<Integer> finished = new ArrayList<>();

        for (int p : V) {
            if (!visited[p]) {
                String result = finishTime(V, E, p, visited, finished);
                if (!result.equals("Inconsistent")) {
                    return reverse(finished);
                }
            }
        }
    }

    public static String finishTime(V, E, int current,
    boolean[] visited, List<Integer> finished) {
        visited[current] = true;
        List<Integer> neighbors = E.get(current);

        for (int v : neighbors) {
            if (visited[v] && !finished.contains(v)) {
                return "Inconsistent";
            }
            else if (!visited[v]) {
                String result = finishTime(V, E, v, visited, finished);
                if (result.equals("Inconsistent")) {
                    return "Inconsistent";
                }
            }
        }
        finished.add(current);
        return "";
    }
}
```

The time complexity for creating this graph is $O(2n)$. We need to add as many as $2 \cdot n$ vertexes because we have to create a birth vertex and death vertex for each person.

The time complexity for DFS should be $O(2n + E)$, where E is the number of edges.

Thus, the total time complexity is $O(4n + E)$.